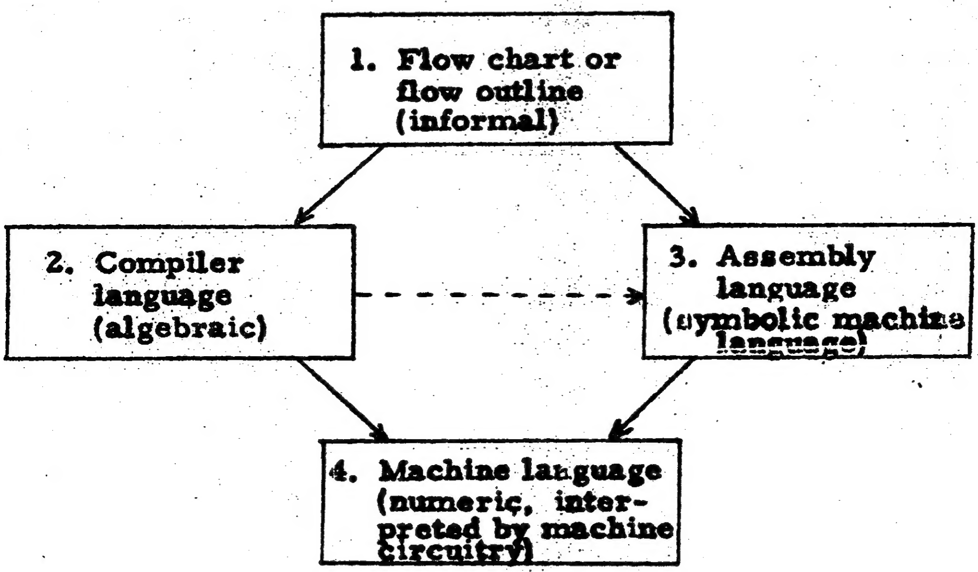
K17a

SEMINAR SERIES ON COMPUTER APPLICATIONS TO BIOLOGY

COMPUTER LANGUAGES by Donald E. Knuth February 8, 1963

The purpose of this lecture is to present, by means of examples, various levels of language used with digital computer programs, and also to give an introduction to programming.

The four principal types of language we will discuss are depicted in this chart:



1. The flow chart or flow outline is a step-by-step description of the procedure to be followed. This is just an aid to the person preparing a program; most programmers like to get their thoughts in order by first preparing a flow chart which shows how to carry out the desired solution. Flow charts have a wide range of levels, depending on the programmer's taste; he may wish to give a brief overall picture of the "flow" of a program,

or he may wish to make an extremely detailed flow description. Language used in flow charts is usually a combination of English and mathematics; in general, the language is sufficiently precise for the programmer to understand, but is far too vague for a computer to understand (at least until we build a computer that thinks). In other words, it takes common sense to determine the meaning of a flow chart description of a program. This is not true for the other languages we describe; they will conform to specific rules, and every statement will have a precise meaning, understandable by the computer.

2. Compiler language is a precise rendering of the flow chart into a well-defined language, which still remains sufficiently close to English and mathematical language to make it easily readable for a non-specialist (after a short briefing). There are dozens of different compiler languages, of which the most commonly used are ALGOL and FORTRAN. (Computer programmers and manufacturers are fond of using acronyms for naming their programs; ALGOL stands for ALGOrithmic Language, FORTRAN stands for FORmula TRANslator.) At CalTech, the ALGOL compiler language is presently used for programs on the Burroughs 220 computer. FORTRAN, which was actually the first compiler language ever designed, still survives today and is presently used for programs on the IBM 7090 computer. The example compiler language to be given in this lecture will be FORTRAN, and you will be getting several lectures on FORTRAN programming in the next few weeks.

3, 4 The machine language of a computer is the code in which instructions are stored inside the machine. The computer automatically analyzes this code and carries out the instructions mechanically. People find it harder to analyze such code, but machine circuitry naturally finds

numerically coded information easy to recognize. Assembly language is directly related to machine language, except it is designed for human comprehension. The programmer does not have to remember many of the details of machine language code; these are abbreviated into a symbolic language, closely related to the way people actually think about the machine instructions. This makes the creation of machine language programs in the equivalent assembly language rather easy. Another important advantage is the programmer's freedom from worrying about clerical details and, consequently, he makes less errors. The programmer who uses an assembly language must still, of course, be intimately familiar with the machine characteristics, but a trained programmer can often dash off an assembly language program as quickly as a flow chart. In this lecture we will discuss FAP (Fortran Assembly Program), an assembly language used for the IBM 7090. A quite different kind of assembly language, which apparently has never been given a name, is now being used here on the Burroughs 220; it is numeric rather than alphabetic and somewhat more closely related to machine language because of the relative lack of equipment for entering alphabetic information into the 220.

Flow charts and compiler languages are customarily called problem-oriented languages, and assembly or machine languages are known as processor-oriented or machine-oriented languages. The big advantage of problem-oriented languages, as far as most research workers are concerned, is that it is for the most part unnecessary for the programmer to learn the details about any specific computer, he can write up programs in a fairly familiar way with little extra training.

Specially written computer programs (called compilers) take anything written in a compiler language and automatically translate it into machine language; then the machine executes the resulting machine language program.

Assembly languages are valuable to programmers who are trained in the intricacies of machine language, as mentioned earlier; other specially written computer programs (called assemblers) translate assembly language into machine language. In fact many compilers translate from the compiler language into assembly language (as indicated by the broken line in the diagram), and an assembler finishes the job.

There is bound to be some loss of efficiency resulting from the automatic translation of compiler language into machine language. However, the programs produced run 90-95% as fast as those written in assembly language by a trained programmer. In fact, compiler-produced programs tend to beat hand-coded problems written by an average programmer when the problem is complex and lengthy.

There are, on the other hand, classes of problems for which compiler output compares poorly with hand-written programs; this is due to a number of features of the machine which are never utilized by the output of a compiler, because of the nature of compiler language. Problems which make heavy use of such facilities are much better when hand-written. (For some combinatorial problems, for example, the compiler programs are less than one per cent efficient! It is to be emphasized, however, that such problems are definitely in the minority.)

1. Flow Diagrams

The problem we will investigate here is a simple everyday problem of making exact change for a given amount of money, using the fewest possible coins in the process. We will assume that the amount to be paid is less than $10.00, and that we have enough coins of each kind to make up any such amount. (The actual assumption is that we have at least one $5 bill, four $1 bills, one half-dollar, one quarter, two dimes, a nickel, four pennies.) A simple method to use for this problem is to find the largest coin less than or equal to the amount, to take one of this coin, and repeat the process with the remaining amount, until the total is reached.

A more explicit way to state this algorithm is in terms of the flow chart below. For communicating an algorithm to other people, I personally tend to favor a flow chart without much detail inside the boxes, accompanied by a flow outline, a set of steps explaining not only what is to be done in the algorithm but why it is done. It is a good idea to prepare such a detailed description so that one can remember several months later what he has done.

Algorithm for making change

S1. An AMOUNT is input for which we are to make up change.

S2. Set J = 1. J will later specify which coin is currently being tried. The coins are (in terms of cents) 500, 100, 50, 25, 10, 5, 1 in that order. Also set K = 0. K equals the number of coins accumulated so far as change.

S3. If AMOUNT is less than the J-th coin, the J-th coin is too big so we go to step S4. If AMOUNT is greater than or equal to the J-th coin, go to step S5.

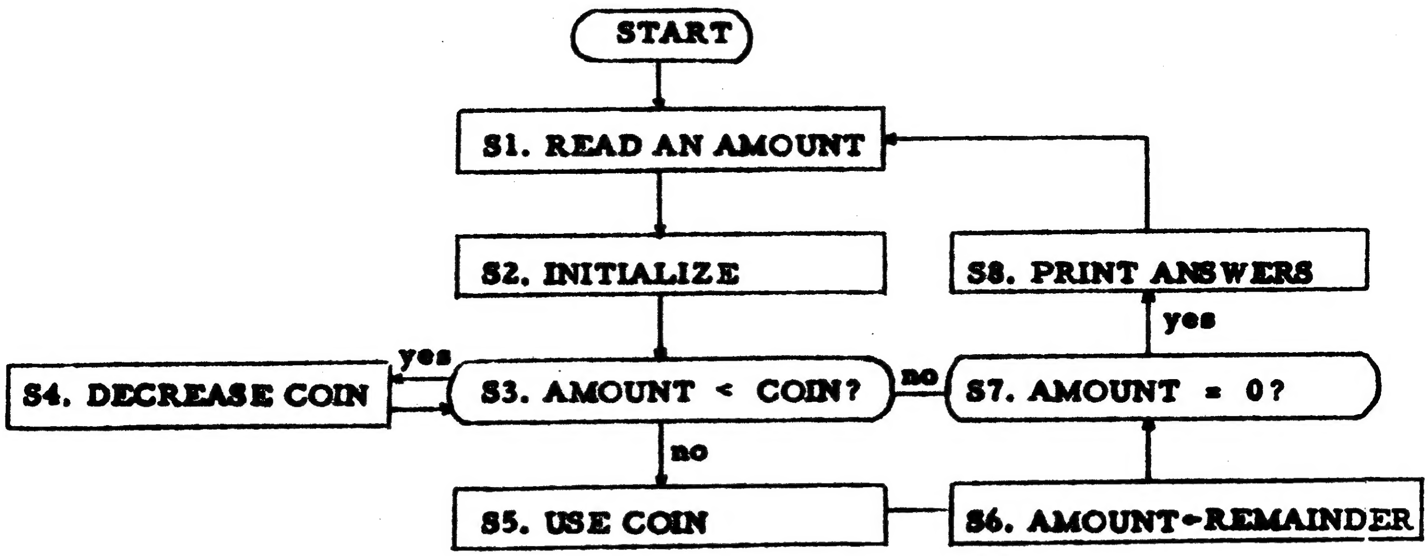
S4. Increase J by 1 (thus specifying the next smaller coin) and return to step S3.

S5. We take one of the J-th ccins as part of the change. This means increase K by one, and set the K-th item of change to the J-th coin.

S6. Decrease AMOUNT by the value of the J-th coin; this represents the amount we must still make up.

S7. If AMOUNT is zero, we are done, so we go on to step S8, otherwise go back to step S3.

S8. Print out the answers for this case, then return to step S1.



II. Compiler Language

Compiler language is quite similar to a flow description; the main differences are that compiler language has a specified form for each kind of operation, and that things are usually adjusted so they are

all on one line. Instead of writing $\frac{A}{B}$ one writes A/B; instead of writing X_2 one writes X(2); instead of writing X^2 one writes X**2; instead of writing $\sqrt{X}$ one writes SQRTF(X). Further explanations of the meaning of each line of the FORTRAN program given below (each line is a so-called "FORTRAN statement") appears together with the statements.

FORTRAN Statements	Explanatory Notes
C MAKING CHANGE	The "C" means this is a comment only.
DIMENSION COIN (7), x CHANGE (13)	This says there are seven coins called COIN (1), COIN (2), . . . COIN (7), and that there are at most thirteen items of change, called CHANGE (1), CHANGE (2), . . . CHANGE (13).
COIN (1) = 500 COIN (2) = 100 COIN (3) = 50 COIN (4) = 25 COIN (5) = 10 COIN (6) = 5 COIN (7) = 1	This series of statements sets up the appropriate values of the coins.
1 READ INPUT TAPE 5, 100, x AMOUNT	This causes a unit of information from the input tape to enter the computer; this record is prepared in format number 100 (see format statements below); the data item is stored in AMOUNT. If no more data is present, the computer terminates the program.
TOTAL = AMOUNT	Set the variable TOTAL equal to the value of the variable AMOUNT. This is done so that when later printing out the answers we remember the original amount.

2 J = 1 K = 0	The variables J and K are set respectively to 1 and 0.
3 IF (AMOUNT - COIN (J)) X 4, 5, 5	The quantity AMOUNT-COIN(J) is computed. If it is negative, go to statement number 4; if it is zero, go to statement number 5; if it is positive, go to statement number 5. (Compare with step S3 of the flow outline.)
4 J = J + 1 GO TO 3	Increase J by 1 Return to step 3
5 K = K + 1 CHANGE (K) = COIN (J)	Increase K by 1 Set CHANGE(K) equal to COIN(J) (Compare with step S5)
6 AMOUNT = AMOUNT - X COIN (J)	Decrease AMOUNT by COIN(J)
7 IF (AMOUNT) 3, 8, 3	If AMOUNT is now negative or positive (i.e. non-zero), return to step 3; if AMOUNT is zero, go to step 8.
8 WRITE OUTPUT TAPE 6, X 200, TOTAL, (CHANGE(I), X I = 1, K)	Record onto magnetic tape a unit of information containing the original TOTAL amount, and the values of CHANGE(1) through CHANGE(K), using format number 200.
GO TO 1	
100 FORMAT (F8. 0)	This is a special notation for the format in which the input cards are punched. (Note: the cards are first key-punched, then transferred to magnetic tape before entering the computer.) The code F8. 0 means eight card columns are used, and the numbers are given without any decimal places.

200 FORMAT (8H0AMOUNT=, x -2P, F7.2, 7H, COINS, 13F7.2)	This format for the answers says to double space, to print titles in the form "AMOUNT= x.xx, COINS", to convert from cents to dollars for readability, and to give up to 13 answers.
END	This indicates the end of the program.

III. Assembly Language

It is not appropriate to explain very much about the internal machine operations in this lecture; a little bit of explanation should be enough to give the basic ideas involved. The IBM 7090 is designed to execute instructions in sequence, and these instructions manipulate numbers. Over 32,000 numbers are kept in the "memory" of the computer. There is a special part of the machine's memory called its registers. The registers that concern us here are the AC register, the MQ register, and the so-called index registers 1 and 2. We will keep the values of TOTAL, COIN, and CHANGE in the ordinary memory, but we will make special assignments as follows:

J is kept in index register 1.

K is kept in index register 2.

AMOUNT is kept in the AC register.

Now here is the same change-making algorithm, expressed in the FAP assembly language. To avoid going into the details of input and output, we will give here only those steps between the "READ INPUT TAPE" and the "WRITE OUTPUT TAPE" parts of our program. An instruction in FAP has four main parts:

NAME OP ADDRESS, INDEX

NAME is the optional symbolic name of the instruction (if the programmer wants to name it); OP specifies the operation to be performed by the machine; ADDRESS specifies on what quantity the operator is to be performed; and INDEX, if it appears, specifies an alteration of the ADDRESS by an index register. Each instruction will be translated into English so that the actual meaning of each line of the code is clear:

NAME	OP	ADDRESS, INDEX	REMARKS
	STO	TOTAL	At the beginning we assume the initial AMOUNT is given to us in the AC register. "STO", in general, means store the AC register into the location given by the address. Therefore, this instruction says, "Store the contents of the AC register into the memory location called TOTAL." It corresponds exactly to the FORTRAN statement in the preceding section TOTAL = AMOUNT, since AMOUNT is in the AC register. The contents of the AC register are not destroyed by the STO instruction.
	AXT	1, 1	"Set index register 1 equal to 1." This corresponds to the FORTRAN statement J = 1, because J is kept in index register 1 here.
	AXT	0, 2	"Set index register 2 equal to 0." (Compare with the preceding instruction.) This corresponds to the statement K = 0 in FORTRAN.
CMPARE	CAS	COIN, 1	This is by far the trickiest instruction of the lot; it is named CMPARE.

The precise meaning is, "Compare the contents of the AC register with the number in the memory location called COIN modified by index register 1 ". This can be translated in our case into "Compare AMOUNT with the J-th coin". If the result of the comparison is greater, the computer goes on to the next instruction. If the result of the comparison is equal, it skips the next instruction and goes on to the following one. Finally if the result of the comparison is less, it skips two instructions and takes the third.

TXI USE, 2, 1

We get to this instruction if AMOUNT is greater than the J-th coin, as we mentioned in the discussion of the previous instruction. This command means, "Increase index register 2 by 1, and the next instruction to execute is the one called USE". In other words, K = K + 1, and we skip to the step named USE below.

TXI USE, 2, 1

We get to this instruction if AMOUNT is exactly equal to the J-th coin. It is exactly the same instruction as the previous.

TXI CMPARE, 1, 1

This instruction reads, "Increase index register 1 by 1, and the next instruction to execute is the one called CMPARE". Note the analogy between this and the preceding; it performs step 3 of our algorithm for us.

USE	LDQ COIN, 1	Here is the instruction called USE. It says, "Load up the contents of the MQ register with the number in memory location COIN modified by index register 1". In other words, the value of the J-th coin is copied into the MQ register.
	STQ CHANGE, 2	"Store the contents of the MQ register in the memory location called CHANGE, modified by index register 2". K is index register 2, so the effect of the last two instructions is to record the value of the J-th coin as the K-th piece of change; CHANGE(K) = COIN(J). Compare this instruction STQ with the first instruction STO. The difference is only in the change from AC to MQ register.
	FSB COIN, 1	"Subtract the value of the J-th coin from the AC register, leaving the result in the AC register". This instruction performs the function AMOUNT = AMOUNT - COIN(J) of the FORTRAN program.
	TNZ CMPARE	"If the contents of the AC register are not zero, the next instruction to execute is the one called CMPARE; otherwise continue on to the next instruction". Thus if the remaining AMOUNT is not zero we continue with step 3 of our procedure, otherwise we have computed all the change as required.

The next instruction, not given here, would then be the beginning of the steps for writing the answers onto the output tape. This completes our example of an assembly language program.

IV. Machine Language

The IBM 7090 internally deals with numbers expressed in the binary system; each ordinary memory location contains a number with 35 binary digits and a plus-or-minus sign. Numbers in the binary system can be easily thought of as numbers in the octal (base 8) number system, by simply grouping binary digits together three at a time; for example, the binary number

+ 00 110 000 001 000 000 000 000 010 000 000 000

(which is the actual way the first instruction, below, appears inside the computer) is easily expressed as the octal number

+ 0 6 0 1 0 0 0 0 2 0 0 0,

and for convenience we will use the octal form.

The memory positions inside the computer are specified by number, using the octal numbers 00000 through 77777. The instructions of a program are coded as numbers, and they are put into the memory along with the numbers manipulated by the program. We will assume the instructions of our program are stored in ascending locations starting with 01000. We also assume that

TOTAL is memory location 02000

COIN is memory location 03000

CHANGE is memory location 04000.

The FAP language program given in part 3 corresponds 1-for-1 with the machine language. For example, the operation STO is written +0601 in

machine language. There are five parts to the machine language instructions: the memory location where it is stored is one, then the contents of that location have four parts, the OP part (as in FAP), the D part (special use depending on the OP), the T part (corresponding to the INDEX in FAP), and finally the ADDRESS part (as in FAP). The reader should compare the address parts given with the corresponding FAP address; the FAP program is reproduced here, copied from part III.

FAP Language			Machine Language				
NAME	OP	ADDRESS, INDEX	Location	OP	D	T	ADDRESS
	STO	TOTAL, 1	01000	+0601	00	0	02000
	AXT	1, 1	01001	+0774	00	1	00001
	AXT	0, 2	01002	+0774	00	2	00000
CMPARE							
	CAS	COIN, 1	01003	+0340	00	1	03000
	TXI	USE, 2, 1	01004	+1000	01	2	01007
	TXI	USE, 2, 1	01005	+1000	01	2	01007
	TXI	CMPARE, 1, 1	01006	+1000	01	1	01003
USE	LDQ	COIN, 1	01007	+0560	00	1	03000
	STQ	CHANGE, 2	01010	-0600	00	2	04000
	FSB	COIN, 1	01011	+0302	00	1	03000
	TNZ	CMPARE	01012	-0100	00	0	01000

Exercises

1. Rewrite the FORTRAN algorithm so as to allow $2 bills in the change.
2. Would our algorithm always produce the least number of coins, if we were not allowed to give nickels? (Hint: consider the case of 30¢.)

```
EXECLTICA       C16145
ANCLAT=    C.25, CCINS    C.25
ANCLAT=    C.26, CCINS    0.25   C.C1
ANCLAT=    C.40, CCINS    C.25   C.10   0.05
ANCLAT=    C.93, CCINS    C.50   C.25   C.10   C.C5   0.01   0.01   0.01
ANCLAT=    0.55, CCINS    C.50   C.25   C.10   C.10
ANCLAT=    9.59, CCINS    5.00   1.C0   1.00   1.00   1.00   0.50   0.25   0.10   0.10   0.01   0.01   0.01   0.01
```